TRANSACTIONAL EXECUTION: TOWARD RELIABLE, HIGH-PERFORMANCE MULTITHREADING

ALTHOUGH LOCK-BASED CRITICAL SECTIONS ARE THE SYNCHRONIZATION METHOD OF CHOICE, THEY HAVE SIGNIFICANT PERFORMANCE LIMITATIONS AND LACK CERTAIN PROPERTIES, SUCH AS FAILURE ATOMICITY AND STABILITY. ADDRESSING BOTH THESE LIMITATIONS REQUIRES CONSIDERABLE SOFTWARE OVERHEAD. TRANSACTIONAL LOCK REMOVAL CAN DYNAMICALLY ELIMINATE SYNCHRONIZATION OPERATIONS AND ACHIEVE TRANSPARENT TRANSACTIONAL EXECUTION BY TREATING LOCK-BASED CRITICAL SECTIONS AS LOCK-FREE OPTIMISTIC TRANSACTIONS.

Ravi Rajwar
Intel Microarchitecture Research Lab

James Goodman
University of Auckland

•••••• Explicit hardware support for multithreaded software, either in the form of shared-memory chip multiprocessors or hardware multithreaded architectures, is becoming increasingly common. As such support becomes available, application developers are expected to exploit these developments by employing multithreaded programming. But although threads simplify the program's conceptual design, they also increase programming complexity. In writing shared-memory multithreaded applications, programmers must ensure that threads interact correctly, and this requires care and expertise. Errors in accessing shared-data objects can cause incorrect program execution and can be extremely subtle.

As hardware support for shared-memory applications, we have proposed the use of Transactional Lock Removal (TLR),[1] which ensures high performance while simplifying error-free programming and providing application stability. TLR uses Speculative Lock Elision (SLE)[2] as an enabling mechanism. In SLE, the hardware dynamically identifies synchronization operations, predicts them as being unnecessary, and elides them. By removing these operations, the program behaves as if synchronization were not present. The hardware can then exploit situations that do not require synchronization for a correct execution, and the hardware makes this determination without even performing the synchronization itself. TLR treats a lock-based critical section as a *lock-free, optimistic transaction*, using SLE to elide the lock, which in turn lets multiple threads enter a critical section protected by the same lock and execute the critical section speculatively. As in SLE, the hardware reads and monitors the lock in a shared state, but ordinarily does not write it. TLR uses the critical shared data itself to order any conflicting data accesses and thus provides successful lock-free execution, *even*

Why Transactions?

Transactions provide an intuitive model for reasoning about coordinated access to shared data.[1] A transaction comprises a series of read and write operations that provide the properties of failure atomicity, consistency, and durability.[2] *Failure atomicity* assures that a transaction either must execute to completion or, in the presence of failures, must appear not to have executed at all. *Consistency* requires the transaction to follow a protocol that provides threads with a consistent view of the data object. Serializability is an intuitive and popular consistency criterion for transactions. Serializability requires the result of executing concurrent transactions to be as if there had been some global order in which these transactions had executed serially. *Durability* states that a committed transaction cannot be undone.

A lack of general transaction support in processors has led to programmers' often relying on critical sections to achieve some of the transactions' functionality. Critical sections are software constructs that enforce mutually exclusive access among threads to shared objects and thus trivially satisfy serializability. Programmers and compilers most commonly implement critical sections with a software construct known as a *lock*. A lock is associated with a shared object and determines if the shared object is currently available. A thread acquires the lock, executes its critical section, and releases the lock. All other threads wait for the lock until the first thread has completed its critical section, serializing access and thus making the entire critical section appear to execute atomically. This waiting prevents a data conflict—in which at least one thread is writing a memory location while at least one other thread is simultaneously reading or writing the same location.

Failure atomicity is difficult to achieve with critical sections, however, because it requires support for logging all modifications performed within a critical section and then making these modifications visible instantaneously using an atomic operation. A transaction, on the other hand, has this property and is thus semantically stronger than a critical section.

References

1. J. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. Int'l Conf. Very Large Databases*, Morgan Kaufmann, 1981, pp. 144-154.
2. K.P. Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, vol. 19, no. 11, Nov. 1976, pp. 624-633.

in the presence of data conflicts. In addition, TLR automatically provides transactional semantics to a critical section.

In this article, we discuss the main limitations of existing programming models, briefly survey related work, and describe key aspects of TLR and its enabling SLE mechanism.

Lock limitations

Nearly all architectures support instructions for implementing lock operations. Locks have become the programmer's synchronization mechanism of choice and are prevalent in operating systems, database servers, and Web servers, among other applications. As the "Why Transactions?" sidebar describes, however, lock-based critical sections have their drawbacks in the presence of unexpected conditions. Two key limitation areas are the programmability-performance tradeoff and application stability.

Programmability-performance tradeoff

The tradeoff between programmability and performance is complex because programmers must reason about data sharing during code development using static, rather than dynamic (runtime), information. Although conservative synchronization can guarantee correctness and lead to quick code development, it also inhibits parallelism because threads serialize unnecessarily. Fine-grained locks (one lock per data-structure field) may help performance, but they can result in code that is both error prone and difficult to write. Coarse-grained locks (one lock per data structure) makes it easier to write correct code, but the attendant locking overhead and serial execution hurt performance.

Synchronization operations, such as locks, date to the 1960s, when the IBM System/360 architecture implemented the first synchronization primitive, test&set (T&S).[3] T&S atomically writes a known value to a memory location, making it possible to test the value of one of the bits immediately before the write occurred. Since then, most research in synchronization has focused on either optimizing the synchronization operation itself or in tolerating the serialization that synchronization causes.

T&S performs well when there is no lock contention, but is inefficient when many threads compete for the lock. The test&test&set (T&T&S) primitive[4] attempts to improve on T&S by having waiting processors spin on a local copy of the lock. Traffic is still substantial with lock contention, however, because again multiple threads compete for the lock whenever the lock release occurs.

Queue-based locking mechanisms attempt to reduce the traffic from lock contention. In this approach, the system maintains a queue of waiting requesters, in which each node typically records pointers to adjacent processors in the queue. The approach minimizes network traffic by having a lock contender perform arbitration for the lock at the time of request, by allowing the processor to spin on a local variable, and by limiting the number of processors involved in the lock transfer. QOLB (queue on lock bit) was the first queue-based locking primitive proposed and required both hardware and software support.[5] Tom Anderson subsequently proposed an all-software queued lock,[6] which John Mellor-Crummey and Michael Scott's MCS lock improved on.[7] Although MCS performs well under contention, it introduces significant software overhead for queue maintenance, which degrades performance in the absence of contention.

QOLB's collocation of lock and data overlaps data transfer with lock transfer, thus reducing latencies observed within critical sections.[5] Speculative execution of critical sections also aims to overlap certain lock-acquisition latencies with computation in the critical section to tolerate lock serialization.[8] Speculative execution works well without contention but degrades performance significantly when there is lock contention because of network traffic on the lock as well as competition for the data among multiple threads. Another technique, Speculative Synchronization,[9] applies thread-level speculation (TLS) to synchronization. In this technique, multiple threads compete for a lock in parallel with the critical section's speculative execution. Without contention, the behavior is similar to that of conventional locking, in that each thread acquires the free lock. With contention, the threads actively compete for the lock while executing their critical sections. Without data conflicts, the TLS-based Speculative Synchronization mechanisms achieve some overlap in critical-section execution among various threads. With data conflicts, threads actively compete for the lock, and at least one thread acquires it. An adaptive version of Speculative Synchronization incorporates an SLE-like mechanism[2] to handle the cases in which data conflicts do not occur. Unlike TLR, neither the resulting adaptive scheme nor the underlying SLE method it employs is lock-free.

Thus, the key performance limitation of synchronization stems from the coordination mechanism among threads and the excessive traffic generated from inefficiently coordinating actively competing threads.[10]

Application stability

In addition to the programmability-performance tradeoff, the software waiting that is characteristic of a lock construct has implications for thread behavior under unexpected conditions, such as thread-scheduling events and failures. If some thread owns a lock by marking it held, other threads requiring that lock must wait for the lock to become free. This waiting can negatively affect system behavior. If the operating system deschedules the lock owner, other threads waiting for the lock cannot proceed because the lock is not free. In highly concurrent environments, all threads may wait until the descheduled thread runs again. However, a nonblocking property requires some thread to make progress even if thread delays or failures occur. Further, if the lock owner aborts, other threads waiting for the lock never complete, since the lock is never free again. The shared structures the aborted thread updated remain in an inconsistent state because critical sections lack failure atomicity. A wait-free property requires all nonfailing threads to complete all operations even with such failures.[11] Conventional locks are neither nonblocking nor wait-free.

Transactional Memory[12] and Oklahoma Update[13] are hybrid mechanisms that attempt to address lock limitations by providing special instructions for constructing concurrent data structures without requiring locks. Although these approaches have good stability properties, they require new instructions, and programmers must learn to use these instructions and reason about the correctness of the resulting data structure. This learning curve limits the appeal of these approaches.

Rethinking the locking problem

Even with their limitations, locks continue to be popular because few alternatives are competitive, and critical sections have intuitive appeal. Lock-based critical sections are nearly universal as mechanisms for synchronizing thread accesses. Even so, lock limitations are becoming notable bottlenecks. We believe that research should pursue solutions for exploiting hardware thread parallelism easily and efficiently. Efficient synchronization must eliminate overhead rather than try to tolerate it. Ideally, locks should be passive, not an object of competition. Serialization should occur only if a data conflict requires it, and then it should occur on the data itself, not on the lock. Given the widespread use of critical sections, any solution to the locking problem would do well to maintain the programmer's model of critical sections. Further, it should address the issues of reliability, performance, and programmability in a unified manner, not through piecemeal mechanisms.

With these points in mind, the ideal goal is to present the programmer with the model of a lock as a zero-overhead synchronization tool. This is exactly the model that TLR provides.[1] It uses modest hardware to convert lock-based critical sections transparently and dynamically into lock-free optimistic transactions and uses fair conflict resolution to provide transactional semantics and starvation freedom. Providing the lowly critical section with transactional execution behavior elevates the simple notion of a critical section to the much more powerful concept of a transaction, yet still uses the familiar acquire/release mechanism. TLR uses SLE[2] as an enabling mechanism to provide a comprehensive solution to the locking problem. TLR's lock-free aspect eliminates all serialization on locks, whether or not there are data conflicts, and provides transparent transactional execution behavior for critical sections.

Speculative lock elision

The aim of SLE is to elide lock acquires from a dynamic execution stream, thus breaking a critical performance barrier by allowing nonconflicting critical sections to execute and commit concurrently. SLE demonstrated for the first time that without data conflicts the hardware can concurrently execute critical sections protected by the same lock and that all these executions can concurrently commit. This is possible because the thread does not have to acquire a lock: The lock need be only *readable* in the processor's cache. Thus, multiple threads can concurrently execute critical sections protected by the same lock and without any dependence on the lock.

The key insight is that locks need not be acquired, only observed. Semantically, the lock is a control variable employed to provide the illusion of atomicity (by actually enforcing mutual exclusion). Thus, removing the lock variable is acceptable if we can use other means to provide the illusion of atomicity.

SLE elides locks without requiring precise semantic information from the software. It enables safe, dynamic lock elision by exploiting a property of locks and critical sections as programmers and compilers commonly implement them. If memory operations between the lock acquire and release appear to occur atomically, elision of the two corresponding writes is possible because the second write (of the lock release) undoes the changes of the first write (of the lock acquire). In other words, the hardware need not perform writes to the lock. To detect atomicity violations, we can use cache-coherence protocols that most modern processors already implement, and employ a rollback mechanism for recovery. On a recovery, the hardware can explicitly write to the lock. Successful elision is validated and committed without acquiring the lock.

SLE works with the speculative-execution hardware available in modern processors. During lock elision, SLE provides the ability to buffer speculative state and recover to an earlier execution point in the event of a misspeculation. Similar to other techniques that use speculative execution, SLE relies on the hardware's ability to buffer speculative state. Thus, when local buffering space is exhausted, speculative execution is not possible—a limitation fundamental to all speculative-execution schemes. However, such a buffering requirement is largely an engineering tradeoff, and processor designers can provide sufficient buffer space to cover nearly all common-case critical sections. Another limitation of speculative-execution techniques, which SLE inherits, is the inability to undo I/O operations. Thus, speculative execution cannot be applied to I/O operations.

SLE requires neither instruction set changes nor programmer or compiler support. Hardware designers can incorporate SLE into modern processors with modest support and without system-level modifications. Further, if any situation arises that precludes SLE application—such as resource constraints or repeated data conflicts—the thread can always acquire the lock normally. Thus, SLE guarantees a correct execution, with the same forward-progress properties as the underlying synchronization mechanism.

Transactional lock removal

TLR aims to achieve a *serializable* schedule of critical sections, in which all memory operations within a critical section are atomically inserted into some global order, independent of data conflicts. Figure 1 illustrates. Serializability requires the result of executing concurrent transactions to be as if these transactions executed in *some* serial order. In the absence of data conflicts, we can ensure serializability using a technique such as SLE but the presence of data conflicts among concurrently executing threads requires additional mechanisms, which TLR provides. TLR performs *active* concurrency control to ensure correct coordinated access to the data that is experiencing conflicting access by using the data itself rather than locks.

TLR is based on four main ideas:

- Locks define a transaction's scope.
- The hardware executes a transaction speculatively without the need for a lock request or acquire.
- A conflict resolution scheme orders conflicting transactions.
- An enabling technique such as SLE gives the appearance that the transaction has atomically committed.

In SLE, a data conflict among concurrent critical sections can result in the threads restarting and using the underlying lock-based scheme. In contrast, TLR uses an explicit data-conflict-resolution scheme to determine which thread need not restart. The conflict resolution allows the winning thread to retain ownership of a conflicting data block. By uniformly applying such a scheme, we ensure that one thread eventually wins all conflicts and retains ownership of all conflicting data blocks. Guaranteeing a winner among all concurrent conflicting threads in turn means that threads can avoid acquiring a lock even in the presence of conflicts. The thread, which wins all conflicts and completes its critical section, makes all its updates visible to other threads instantaneously at the end of the critical section. Designers can implement the conflict resolution scheme in hardware using existing cache-coherence protocols.[1]

As in any speculative-execution scheme, TLR works as long as the speculative state is buffered. In such cases, all nonconflicting transactions proceed and complete concurrently without serialization or lock dependence. The processor also executes critical

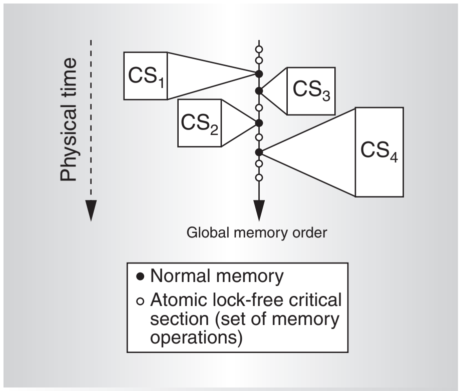

Figure 1. TLR conceptual effect. Although critical-section executions without lock acquires (CS_i) overlap in physical time (with or without data conflicts), each critical section logically appears to be inserted atomically and instantly in a global ordering or memory operations.

sections experiencing data conflicts without the need to acquire locks and without affecting critical sections that do not have data conflicts. The decision of when to serialize execution is based on the accessed data experiencing a conflict, rather than on a competition for the lock. Further, TLR provides failure atomicity, even with repeated data conflicts.

Numerous conflict-resolution schemes are possible, but to provide starvation freedom, we favor a scheme based on timestamps, which we discuss in our original paper on TLR.[1] These timestamps can be viewed as a priority, and all operations within the critical section are assigned the same timestamp. TLR uses timestamps solely to determine which of the two conflicting threads has a higher priority, not to *explicitly* order the execution of critical sections among different processors. Thus with TLR, transactions that conflict in their data sets but do not actually observe any detected conflicts during their execution can execute in *any* order independent of the transactions' timestamps.

TLR performance

To understand TLR performance, we studied many points in the spectrum of behavior possible in the execution of critical sections with data conflicts. Three of these points are particularly illuminating—coarse-grained locking with no data conflicts, fine-grained locking with high data conflicts, and fine-grained

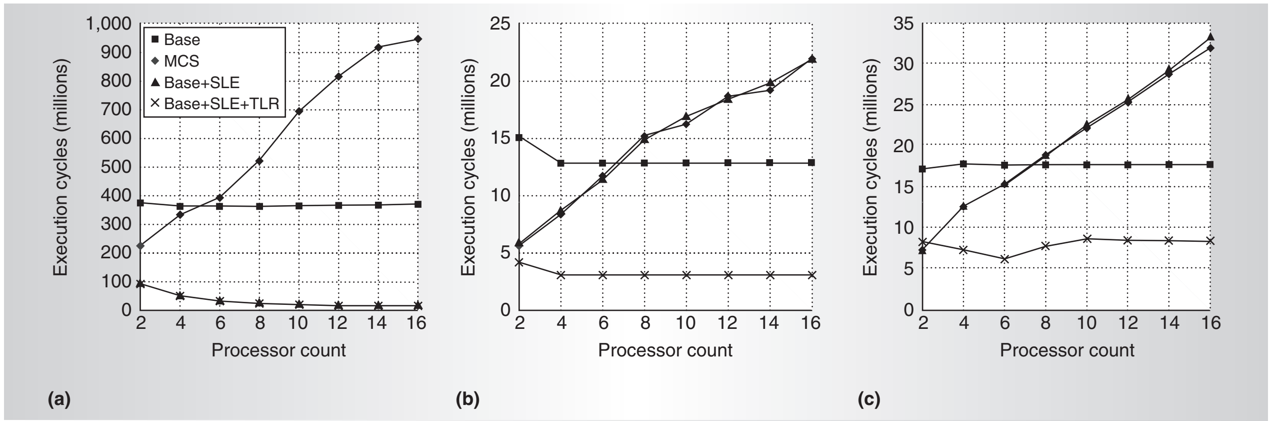

Figure 2. TLR performance for three critical-section behaviors. The multiple-counter microbenchmark corresponds to coarse-grained locking and no conflicts (a). The single-counter benchmark corresponds to fine-grained locking and high conflicts (b). The doubly-linked-list benchmark corresponds to fine-grained locks and dynamic conflicts (c). This figure also shows the performance of test&test&set (Base), MCS locks (MCS), the Base scenario with SLE (Base+SLE), and the Base scenario with SLE and TLR (Base+SLE+TLR).

locking with dynamic data conflicts. The three points correspond to three microbenchmarks.

The *multiple-counter* microbenchmark (coarse-grained locking with no data conflicts) consists of *n* counters protected by a single lock. Each processor uniquely updates only one of *n* counters $2^{24}/n$ times. Even though a single lock protects the counters, there is no dependence across the various critical sections for the data itself and hence no conflicts.

The *single-counter* microbenchmark (fine-grained locking with high data conflicts) corresponds to critical sections operating on a single cache line. A lock protects one counter, and *n* processors increment the counter $2^{16}/n$ times. There is no inherent exploitable parallelism, since all processors operate on the same data (and cache line).

The *doubly-linked-list* microbenchmark (fine-grained locking with dynamic data conflicts) consists of a doubly-linked list with head and tail pointers protected by one lock. Each processor dequeues an item pointed to by the head pointer and enqueues it to the tail up to $2^{16}/n$ times. Concurrent enqueue and dequeue operations can occur on a nonempty queue. However, both head and tail must be modified for an empty queue. Exploiting such concurrency is nontrivial. The critical sections involve pointer manipulations and multiple cache line accesses.

The system we used in this simulation employs T&T&S (Base), software MCS locks (MCS), SLE (Base+SLE), and TLR (Base+SLE+TLR). Figure 2 presents the results. We describe our complete set of results in detail in our original paper on TLR.[1]

As Figure 2a shows, Base performance degrades with more threads because of the severe competition for the lock. The traffic that T&T&S generates overwhelms the network. MCS, as expected, is scalable under high contention but experiences fixed software overhead. TLR and SLE behave identically because there are no data conflicts, and both outperform Base and MCS.

In Figure 2b, Base performance degrades with increasing threads because of severe contention for both the lock and data. SLE behaves similarly to Base because SLE detects frequent data conflicts, turns off speculation, and falls back to the Base scheme. MCS is scalable but experiences fixed software overhead. TLR performs well because no lock conflicts occur.

In Figure 2c, Base performance degrades as it did in Figure 2a and 2b because of severe lock contention. SLE does not perform well either and performs similarly to Base. Determining when to apply speculation is difficult because of this benchmark's dynamic concurrency. SLE often falls back to the base case of lock acquisitions because of detected data conflicts. MCS again is scalable but experiences fixed software overhead. TLR performs well and can exploit enqueue/dequeue concurrency.

In summary, TLR outperforms both Base

and MCS. TLR exploits dynamic concurrency, while synchronization performance limits both Base and MCS. MCS performs a constant factor worse than TLR, while Base performance degrades quite substantially with increasing contention. The behavior of Base under lock contention is poor because multiple processors are racing for the lock and data, resulting in repeated access to the lock variable and the introduction of considerable traffic into the network. That the T&T&S profile remains the same regardless of the workload demonstrates the fundamental performance problems of locking, independent of data characteristics.

Implications of transactional execution

TLR has implications for a wide range of multithreaded applications and related issues,[14] notably programmability and performance, stability, data races, and databases.

Programmability and performance

With transactional execution, there is little need to reason about lock granularity because the hardware makes ordering decisions dynamically on the basis of actual data conflicts and independently of lock granularity. Thus, transactional execution solves a critical problem in reasoning about writing multithreaded programs. TLR extracts and exploits the fine-grained parallelism inherent in the program, regardless of the locking granularity the programmer used. Because the hardware makes serialization decisions only when data conflicts occur and only for threads involved in the conflict, TLR automatically achieves the performance of the finest granularity locking, and the programmer can freely employ coarse-grained locking instead of fine-grained locking without concern for performance or deadlock. Hardware data transfers are efficient and have low overhead. Programmers can thus focus on writing correct code and let the hardware automatically extract performance.

Stability

Eliminating the software wait on locks (because locks are not written) provides lock- and wait-free properties transparently. Consequently, systemwide interactions improve, behavior is nonblocking, and stability increases. Additionally, TLR automatically provides failure atomicity, subject only to resource constraints. Further, TLR provides lightweight support for restartable critical sections—a direct result of SLE's failure atomicity guarantee (in the absence of data conflicts) and TLR's similar guarantee (in the presence of data conflicts). This restartable property is a powerful functionality that applications can exploit.

Data races

TLR's enforced atomicity (and thus transactional execution) ensures that data accesses within critical sections occur atomically and that no access, within or out of other critical sections, from other threads can be interleaved within a critical section. This in effect masks such data races, forcing these racing accesses to be ordered either before or after a critical section. In this way, TLR prevents subtle, undesirable data races for a given execution—a potentially powerful mechanism for reliable, race-free execution. This can also free the programmer from worrying about data races when using transactions, since all operations in a transaction will occur in a race-free manner.

Databases

Four properties of database transactions are atomicity, consistency, isolation, and durability. Atomicity requires that a transaction either execute to completion or have no effects at all. TLR attains atomicity by buffering updates performed within a lock-free critical section and writing them to memory only if the lock-free critical section completes. On a failure, the TLR mechanism discards all updates performed within the failed critical section. A transaction must preserve the consistency of shared data, which is a property of the transaction itself, not of the mechanism that implements it. Isolation requires that transactions execute in a way that allows them to be serialized. TLR meets this requirement by employing an appropriate conflict-resolution scheme. Durability requires that the results of transactions be successfully committed to storage, and this state must be restored in the event of any failure. Because TLR does not handle disk writes, it does not provide durability. Ravi Rajwar and Philip Bernstein discuss TLR in the context of databases,[15] noting that TLR's failure atomicity, consistency, and isolation properties are a promising fit for high-performance database operations.

Toward generalized transactions

TLR's atomic transaction abstraction is a powerful primitive for constructing richer software operations because it lets programmers continue using critical sections without the limitations inherent in locking. Using TLR as a transaction mechanism removes the locking overhead characteristic of the standard lock-based approach. TLR automatically handles lock-based programs that implicitly treat the lock-acquire and lock-release operations as beginning and ending a transaction, but it is also an excellent fit for generalized transactional execution—the lock-acquire operation corresponds to transaction_begin, and the lock-release corresponds to transaction_end. Exposing such a paradigm to the programmer via new constructs such as transaction_begin and transaction_end, while allowing the critical section paradigm to coexist, might hold the key to improved reliability in future multithreaded applications. The programmer focuses on reliability and correctness via transactions, and the hardware achieves high performance for most transactions. For the remaining small fraction that does not lend itself to TLR because of resource issues and I/O, a slower software scheme in the form of a library could be the solution. Such a hybrid approach—TLR for most cases and the slower software scheme for the uncommon case—will provide reliable high performance most of the time and an execution with a continued strong reliability the rest of the time. TLR lets both legacy code and applications based on critical sections benefit from transactional properties, while enabling a future programming model based exclusively on generalized transactions. In this way, it becomes an excellent migration path for reliable programs.

In the transactional execution scheme we propose, SLE provides the mechanism to extract a lock-free execution from a lock-based execution and guarantees such an execution in the absence of data conflicts. TLR uses SLE as an enabling mechanism but, in addition, provides a successful lock-free execution, even in the presence of data conflicts. We believe future software systems should use transactions for improving their reliability and programmability, and hardware mechanisms such as TLR should provide the common-case performance for such software systems. The uncommon case might be handled using a slower software interface, thus guaranteeing transaction properties in all cases. This decoupling of performance and programmability holds the key to future reliable high-performance systems. MICRO

References

1. R. Rajwar and J.R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 5-17.
2. R. Rajwar and J.R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proc. 34th Int'l Symp. Microarchitecture* (MICRO-34), IEEE CS Press, 2001, pp. 294-305.
3. G.M. Amdahl, G.A. Blaauw, and F.P. Brooks Jr., "Architecture of the IBM System/360," *IBM J. Research and Development*, Apr. 1964, pp. 87-101.
4. L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. Int'l Symp. Computer Architecture* (ISCA 84), ACM Press, 1984, pp. 340-347.
5. J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 89), ACM Press, 1989, pp. 64-75.
6. T.E. Anderson, "The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, vol. II (software), IEEE CS Press, 1989, pp. 170-174.
7. J.M. Mellor-Crummey and M.L. Scott, "Synchronization without Contention," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 91), ACM Press, 1991, pp. 269-278.
8. K. Gharachorloo, A. Gupta, and J.L. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. Int'l Conf. Parallel Processing,* IEEE CS Press, 1991, pp. 355-364.

9. J.F. Martínez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 18-29.
10. A. Kägi, D. Burger, and J.R. Goodman, "Efficient Synchronization: Let Them Eat QOLB," *Proc. Int'l Symp. Computer Architecture* (ISCA 97), IEEE CS Press, 1997, pp. 170-180.
11. M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 1, Jan. 1991, pp. 124-129.
12. M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, 1993, pp. 289-300.
13. J.M. Stone et al., "Multiple Reservations and the Oklahoma Update," *IEEE Parallel & Distributed Technology*, vol. 1, no. 6, Nov. 1993, pp. 58-71.
14. R. Rajwar, "Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs," PhD dissertation, CS Dept., Univ. of Wisconsin-Madison, 2002.
15. R. Rajwar and P.A. Bernstein, "Atomic Transactional Execution in Hardware: A New High-Performance Abstraction for Databases," *Int'l Workshop High-Performance Transaction Systems,* 2003 (position paper); http://research.sun.com/hpts2003/.

The biography of **Ravi Rajwar** appears on p. 19 of this issue.

James Goodman is a professor of computer science at the University of Auckland, New Zealand. He is on leave from the University of Wisconsin-Madison, where he performed the work reported in this article. His research interests include memory systems and parallel computing. Goodman has a PhD in electrical engineering and computer science from the University of California, Berkeley.

Direct questions and comments about this article to Ravi Rajwar, Intel Microprocessor Research Lab; ravi.rajwar@intel.com.